Basic Search

Philipp Koehn

13 February 2024
Outline

• Problem-solving agents

• Problem types

• Problem formulation

• Example problems

• Basic search algorithms
problem-solving agents
Problem Solving Agents

Restricted form of general agent:

```
function SIMPLE-PROBLEM-SOLVING-AGENT(percept) returns an action
    static: seq, an action sequence, initially empty
    state, some description of the current world state
    goal, a goal, initially null
    problem, a problem formulation
    state ← UPDATE-STATE(state, percept)
    if seq is empty then
        goal ← FORMULATE-GOAL(state)
        problem ← FORMULATE-PROBLEM(state, goal)
        seq ← SEARCH(problem)
        action ← RECOMMENDATION(seq, state)
        seq ← REMAINDER(seq, state)
    return action
```

Note: this is offline problem solving; solution executed “eyes closed.”

**Online** problem solving involves acting without complete knowledge.
Example: Romania

Diagram showing the network of cities in Romania with distances between them.
Example: Romania

- On holiday in Romania; currently in Arad
- Flight leaves tomorrow from Bucharest
- Formulate goal
  - be in Bucharest
- Formulate problem
  - states: various cities
  - actions: drive between cities
- Find solution
  - sequence of cities, e.g., Arad, Sibiu, Fagaras, Bucharest
problem types
Problem Types

- **Deterministic, fully observable** $\implies$ single-state problem
  - agent knows exactly which state it will be in
  - solution is a sequence

- **Non-observable** $\implies$ conformant problem
  - Agent may have no idea where it is
  - solution (if any) is a sequence

- **Nondeterministic and/or partially observable** $\implies$ contingency problem
  - percepts provide new information about current state
  - solution is a contingent plan or a policy
  - often interleave search, execution

- **Unknown state space** $\implies$ exploration problem (“online”)
Example: Vacuum World

**Single-state**, start in #5. Solution?  
[Right, Suck]

**Conformant**, start in \{1, 2, 3, 4, 5, 6, 7, 8\}  
e.g., Right goes to \{2, 4, 6, 8\}. Solution?  
[Right, Suck, Left, Suck]

**Contingency**, start in #5  
Murphy’s Law: Suck can dirty a clean carpet  
Local sensing: dirt, location only.  
Solution?  
[Right, if dirt then Suck]
problem formulation
A problem is defined by four items:

- **initial state** e.g., “at Arad”

- **successor function** $S(x) = \text{set of action–state pairs}$
  e.g., $S(\text{Arad}) = \{ \langle \text{Arad} \rightarrow \text{Zerind}, \text{Zerind} \rangle, \ldots \}$

- **goal test**, can be
  - **explicit**, e.g., $x = \text{“at Bucharest”}$
  - **implicit**, e.g., $\text{NoDirt}(x)$

- **path cost** (additive)
  e.g., sum of distances, number of actions executed, etc.
  $c(x, a, y)$ is the **step cost**, assumed to be $\geq 0$

A solution is a sequence of actions leading from the initial state to a goal state.
**Example: Vacuum World State Space Graph**

**states?**: integer dirt and robot locations (ignore dirt amounts etc.)

**actions?**: Left, Right, Suck, NoOp

**goal test?**: no dirt

**path cost?**: 1 per action (0 for NoOp)
Example: The 8-Puzzle

- **states?**: integer locations of tiles (ignore intermediate positions)
- **actions?**: move blank left, right, up, down (ignore unjamming etc.)
- **goal test?**: = goal state (given)
- **path cost?**: 1 per move

[Note: optimal solution of $n$-Puzzle family is NP-hard]
Example: Robotic Assembly

states?: real-valued coordinates of robot joint angles
parts of the object to be assembled
actions?: continuous motions of robot joints
goal test?: complete assembly
path cost?: time to execute
tree search
Tree Search Example
Tree Search Example
Tree Search Example
Tree Search Algorithms

- Basic idea: offline, simulated exploration of state space by generating successors of already-explored states (a.k.a. expanding states)

```plaintext
function Tree-Search(problem, strategy) returns a solution, or failure
    initialize the search tree using the initial state of problem
    loop do
        if there are no candidates for expansion then return failure
        choose a leaf node for expansion according to strategy
        if the node contains a goal state then return the corresponding solution
        else expand the node and add the resulting nodes to the search tree
    end
```

Implementation: States vs. Nodes

- A state is a (representation of) a physical configuration.
- A node is a data structure constituting part of a search tree includes parent, children, depth, path cost $g(x)$.
- States do not have parents, children, depth, or path cost!

The EXPAND function creates new nodes, filling in the various fields and using the SUCCESSORFn of the problem to create the corresponding states.
Search Strategies

• A strategy is defined by picking the **order of node expansion**

• Strategies are evaluated along the following dimensions
  
  – **completeness**—does it always find a solution if one exists?
  – **time complexity**—number of nodes generated/expanded
  – **space complexity**—maximum number of nodes in memory
  – **optimality**—does it always find a least-cost solution?

• Time and space complexity are measured in terms of
  
  – $b$ — maximum branching factor of the search tree
  – $d$ — depth of the least-cost solution
  – $m$ — maximum depth of the state space (may be $\infty$)
Uninformed strategies use only the information available in the problem definition

- Breadth-first search
- Uniform-cost search
- Depth-first search
- Depth-limited search
- Iterative deepening search
breadth-first search
Breadth-First Search

- Expand shallowest unexpanded node

**Implementation:**

fringe is a FIFO queue, i.e., new successors go at end

fringe = (A)
Breadth-First Search

- Expand shallowest unexpanded node

**Implementation:**

*fringe* is a FIFO queue, i.e., new successors go at end

\[ \text{fringe} = (B,C) \]
Breadth-First Search

- Expand shallowest unexpanded node

**Implementation:**
*fringe* is a FIFO queue, i.e., new successors go at end

\[ \text{fringe} = (C,D,E) \]
Breadth-First Search

- Expand shallowest unexpanded node

**Implementation:**
- *fringe* is a FIFO queue, i.e., new successors go at end

\[ fringe = (D,E,F,G) \]
Properties of Breadth-First Search

- **Complete?** Yes (if \( b \) is finite)
- **Time?** 
  \[ 1 + b + b^2 + b^3 + \ldots + b^d + b(b^d - 1) = O(b^{d+1}), \text{i.e., exp. in } d \]
- **Space?** \( O(b^{d+1}) \) (keeps every node in memory)
- **Optimal?** Yes (if cost = 1 per step); not optimal in general
- **Space** is the big problem
uniform cost search
Uniform-Cost Search

- Expand least-cost unexpanded node

- **Implementation:**
  \( \text{fringe} = \text{queue ordered by path cost, lowest first} \)

- Equivalent to breadth-first if step costs all equal

- **Properties**
  - Complete? \( \text{Yes, if step cost} \geq \epsilon \)
  - Time? \( \# \text{ of nodes with} g \leq \text{cost of optimal solution}, O(b^{\lceil C^*/\epsilon \rceil}) \)
    where \( C^* \) is the cost of the optimal solution
  - Space? \( \# \text{ of nodes with} g \leq \text{cost of optimal solution}, O(b^{\lceil C^*/\epsilon \rceil}) \)
  - Optimal? \( \text{Yes} \text{— nodes expanded in increasing order of} g(n) \)
depth first search
Depth-First Search

- Expand deepest unexpanded node

Implementation:

fringe = LIFO queue, i.e., put successors at front

fringe = (A)
Depth-First Search

- Expand deepest unexpanded node

- **Implementation:**
  - \textit{fringe} = LIFO queue, i.e., put successors at front

\textit{fringe} = (B,C)
Depth-First Search

- Expand deepest unexpanded node

**Implementation:**

- *fringe* = LIFO queue, i.e., put successors at front

```
fringe = (D,E,C)
```
Depth-First Search

- Expand deepest unexpanded node

- Implementation:
  - fringe = LIFO queue, i.e., put successors at front

\[ \text{fringe} = \{H, I, E, C\} \]
Depth-First Search

- Expand deepest unexpanded node

**Implementation:**

*fringe* = LIFO queue, i.e., put successors at front

*fringe* = (I,E,C)
Depth-First Search

- Expand deepest unexpanded node

- **Implementation:**
  - \textit{fringe} = LIFO queue, i.e., put successors at front

\textit{fringe} = (E,C)
Depth-First Search

- Expand deepest unexpanded node

- **Implementation:**
  
  `fringe` = LIFO queue, i.e., put successors at front

`fringe` = (J,K,C)
Depth-First Search

- Expand deepest unexpanded node

**Implementation:**

fringe = LIFO queue, i.e., put successors at front

fringe = (K,C)
Depth-First Search

- Expand deepest unexpanded node

**Implementation:**

- *fringe* = LIFO queue, i.e., put successors at front

![Diagram](image)

fringe = (C)
Depth-First Search

- Expand deepest unexpanded node

**Implementation:**
- \textit{fringe} = LIFO queue, i.e., put successors at front

\textit{fringe} = (F,G)
Depth-First Search

- Expand deepest unexpanded node

- **Implementation:**
  
  \[ \text{fringe} = \text{LIFO queue, i.e., put successors at front} \]

\[ \text{fringe} = (\text{L}, \text{M}, \text{G}) \]
Depth-First Search

- Expand deepest unexpanded node

- **Implementation:**
  
  \( \text{fringe} = \) LIFO queue, i.e., put successors at front

\( \text{fringe} = (M,G) \)
Properties of Depth-First Search

- **Complete?**
  - no: fails in infinite-depth spaces, spaces with loops
  - modify to avoid repeated states along path
    ⇒ complete in finite spaces

- **Time?** $O(b^m)$
  - terrible if $m$ is much larger than $d$
  - but if solutions are dense, may be much faster than breadth-first

- **Space?** $O(bm)$, i.e., linear space!

- **Optimal?** No
iterative deepening
Depth-Limited Search

- Depth-first search with depth limit \( l \), i.e., nodes at depth \( l \) have no successors
- **Recursive implementation:**

```
function Depth-Limited-Search( problem, limit ) returns soln/fail/cutoff
RECURSIVE-DLS(MAKE-NODE(INITIAL-STATE[problem]), problem, limit)

function RECURSIVE-DLS( node, problem, limit ) returns soln/fail/cutoff
    cutoff-occurred? ← false
    if GOAL-TEST( problem, STATE[node] ) then return node
    else if DEPTH[node] = limit then return cutoff
    else for each successor in EXPAND( node, problem ) do
        result ← RECURSIVE-DLS( successor, problem, limit )
        if result = cutoff then cutoff-occurred? ← true
        else if result ≠ failure then return result
    if cutoff-occurred? then return cutoff else return failure
```
Iterative Deepening Search

function ITERATIVE-DEEPENING-SEARCH(problem) returns a solution

inputs: problem, a problem

for depth ← 0 to ∞ do
    result ← DEPTH-LIMITED-SEARCH(problem, depth)
    if result ≠ cutoff then return result
end
Iterative Deepening Search $l = 0$
Iterative Deepening Search $l = 1$
Iterative Deepening Search $l = 2$
Iterative Deepening Search $l = 3$
Properties of Iterative Deepening Search

- **Complete?** Yes
- **Time?** \((d + 1)b^0 + db^1 + (d - 1)b^2 + \ldots + b^d = O(b^d)\)
- **Space?** \(O(bd)\)
- **Optimal?** Yes, if step cost = 1
  
  Can be modified to explore uniform-cost tree

  - Numerical comparison for \(b = 10\) and \(d = 5\), solution at far right leaf:

  \[
  N(\text{IDS}) = 50 + 400 + 3,000 + 20,000 + 100,000 = 123,450
  \]

  \[
  N(\text{BFS}) = 10 + 100 + 1,000 + 10,000 + 100,000 + 999,999 = 1,111,100
  \]

  - IDS does better because other nodes at depth \(d\) are not expanded
  - BFS can be modified to apply goal test when a node is **generated**
## Summary of Algorithms

<table>
<thead>
<tr>
<th>Criterion</th>
<th>Breadth-First</th>
<th>Uniform-Cost</th>
<th>Depth-First</th>
<th>Depth-Limited</th>
<th>Iterative Deepening</th>
</tr>
</thead>
<tbody>
<tr>
<td>Complete?</td>
<td>Yes*</td>
<td>Yes*</td>
<td>No</td>
<td>Yes, if ( l \geq d )</td>
<td>Yes</td>
</tr>
<tr>
<td>Time</td>
<td>( b^{d+1} )</td>
<td>( b^{C^*/\epsilon} )</td>
<td>( b^m )</td>
<td>( b^l )</td>
<td>( b^d )</td>
</tr>
<tr>
<td>Space</td>
<td>( b^{d+1} )</td>
<td>( b^{C^*/\epsilon} )</td>
<td>( b^m )</td>
<td>( b^l )</td>
<td>( b^d )</td>
</tr>
<tr>
<td>Optimal?</td>
<td>Yes*</td>
<td>Yes</td>
<td>No</td>
<td>No</td>
<td>Yes*</td>
</tr>
</tbody>
</table>

\(|b| = \frac{\text{Cost}}{\epsilon} \approx C^*/\epsilon\)
Repeated States

Failure to detect repeated states can turn a linear problem into an exponential one.
Graph Search

function **GRAPH-SEARCH**(*problem, fringe*) returns a solution, or failure

closed $\leftarrow$ an empty set
fringe $\leftarrow$ INSERT(MAKE-NODE(INITIAL-STATE[*problem*]), fringe)

loop do
  if fringe is empty then return failure
  node $\leftarrow$ REMOVE-FRONT(fringe)
  if GOAL-TEST(*problem, STATE[node]*) then return node
  if STATE[node] is not in closed then
    add STATE[node] to closed
    fringe $\leftarrow$ INSERTALL(EXPAND(node, *problem*), fringe)
end
Summary

• Problem formulation usually requires abstracting away real-world details to define a state space that can feasibly be explored

• Variety of uninformed search strategies

• Iterative deepening search uses only linear space and not much more time than other uninformed algorithms

• Graph search can be exponentially more efficient than tree search